USING A DEBUGGING FRAMEWORK TO ENFORCE BEST PRACTICES IN PROGRAM DEVELOPMENT

BACKGROUND OF THE INVENTION

Field of the Invention

[0001] This invention, generally, relates to debugging computer programs. More specifically, the invention relates to the automatic detection of problematic coding patterns and violations of best practices patterns at program runtime.

Background Art

[0002] In any large software deployment, subtle coding defects can cause problems in successful deployment of the software due to system outages, or incorrect program behavior. If these defects are left unchecked, they may surface as problems only when the program is deployed in the production environment, or when the program is subjected to considerable load.

[0003] Tracking down these defects is extremely difficult in the production environment because of a number of factors. First, the presenting symptoms are usually not unique to a particular type of software defect; several different types of defects can result in the same program behavior at runtime (e.g. an unresponsive system). This fact makes correlating symptoms to specific defects nearly impossible. Second, the presenting symptoms of the defect only manifest themselves during production level loads and production configurations. This means that the defects are often undetected in the testing and debugging of software, and surface under conditions that make the defect tracking process slow, difficult and expensive. Third, the factors that make a piece of code defective are often a complex pattern that may span third party libraries and frameworks, which makes the detection of the defect difficult for an individual developer, unless he or she can gain a global perspective on the software that comprises both the application and the framework layers.

SUMMARY OF THE INVENTION

[0004] An object of this invention is to enable a software developer to track subtle defects in a running program during the development process.

[0005] Another object of the invention is to provide a framework for enforcement of a large number of best practices rules in a running program.

[0006] A further object of the present invention is to detect automatically problematic coding patterns or violations of best practices patterns at program runtime, with very little intervention required by the user.

[0007] These and other objectives are attained with a tool and method for monitoring the behavior of a running computer program. The tool observes the behavior of a running program within the context of a large number of defined coding patterns, and automatically flags violations of the coding patterns when they occur. These defined coding patterns can include best practice patterns and problematic coding patterns. The tool, in the preferred embodiment of the invention, does this using a standard debugger to enforce the coding pattern. This is advantageous because debuggers contain the type of information needed both to enforce the rules and to explain the violations of the rules. Moreover, because debuggers are standard tools in developers' arsenal, little or no additional training is required for the user of the tool to detect subtle software defects.

[0008] Further benefits and advantages of the invention will become apparent from a consideration of the following detailed description, given with reference to the accompanying drawings, which specify and show preferred embodiments of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] Figure 1 illustrates a software architecture and methodology of the tool of this invention deployed within a software integrated development environment (IDE).

[0010] Figure 2 is a more detailed flow diagram that depicts the methodology employed by the tool of this invention.

[0011] Figure 3 is a flow chart showing a preferred procedure for implementing the present invention.

DETAILED DESCRPIPTION OF THE PREFERRED EMBODIMENTS

[0012] The preferred embodiment of this invention provides a tool that detects violations of formalized coding patterns applicable to a specific platform (e.g., J2EE or .NET) in a running program through a debugger. These coding patterns can include best practice patterns and problematic coding patterns.

[0013] Figure 1 is a diagram that depicts a software architecture 10 and methodology of the preferred tool of the invention deployed within a software integrated development environment (IDE). In this embodiment, a developer writes code 12 in an IDE. The developer also typically has test cases 14 to run this code (along with dependent libraries or middleware) within this IDE. The tool 16 provides a mechanism to hook into the debugger 18 within the IDE, such that when the test cases are run by the developer in the debug mode, violations of formalized best practices rules can be automatically detected.

[0014] Figures 2 and 3 are more detailed flow diagrams which depict the methodology employed by the tool 16. The steps and components involved in enforcement of best practice patterns in a running program are discussed below.

[0015] At step 22, the launch of a program within debug mode is intercepted by a component 24 referred to as the launcher, and at step 26, a pattern detector manager 30 is called. The launcher is the component that intercepts the launch, and the pattern detector manager is a component that manages the different pattern detectors 32 that operate within the tool. At step 34, the pattern detector manager automatically inserts entry/exit breakpoints at key methods. This is usually at one or more main entry points into the program (e.g., a program's main method).

[0016] As represented at step 36, the pattern detector manager registers a breakpoint listener for this breakpoint. As represented by step 40, upon hitting the key method entry breakpoint, each pattern detector that is in the control of the pattern detector manager is then instructed by the pattern detector manager to turn on additional breakpoints necessary to identify the first step(s) in identifying their respective defect patters. To set these additional breakpoints, the pattern detector may consult a coding patterns database 42.

[0017] At step 44, each pattern detector tracks these additional breakpoints to detect code defects, and dynamically turns on or off breakpoints based on the breakpoints that are hit for the pattern. When a pattern detector identifies a problematic coding pattern or a violation of a best practice, the violation is, at step 46, reported by reporter 48 as a defect to the developer. A pattern detector can identify a violation either by a series of events happening which should not happen or by expected events being missing when program execution or the trace code segment completes.

[0018] As an example, assume that there are four pattern detectors within the tool 16 that detect violations of the following four types of best practice patterns:

1. *Must call X after a call to Y*, which detects that a method X is called after a call to method Y on the same object.
2. *Never call X from Y*, which detects if a method X was called from method Y either directly or indirectly through a number of intervening method calls.
3. *Do not call X from Y more than Z times*, which detects if a method X was called from method Y directly or indirectly more than Z times.
4. *Do not store objects that are greater than size Z in X through method Y*, which makes sure that the total size of objects that are stored in object X through the method Y is not greater than Z.

[0019] In this example, the launcher 24 intercepts program launch in debug mode, and calls the pattern detector manager 30 to set a key breakpoint for entry into the main method. When the main method entry level breakpoint is hit, each of the pattern detectors is called to set additional pattern specific breakpoints. In the example, each of the pattern detectors sets the following set of breakpoints, and tracks them through program execution.

[0020] The Must call X after a call to Y pattern detector will set breakpoints at both methods X and Y. When method Y is called on an object, the reference to that object is stored into a data structure maintained by the pattern detector and a breakpoint is inserted at method X. When method X is called on an object, the reference to the object is removed from the data structure, and if no references remain in the data structure, the breakpoint on method X is removed. At exit from the key method of the program, any object references left in the data structure of the pattern detector are objects where a call to Y was not followed by a call to X. These objects are then reported to the user as violations of the Must call X after a call to Y best practice.

[0021] The Never call X from Y pattern detector will first place a breakpoint at Y. If the Y breakpoint is hit in the course of program execution, an additional breakpoint will be added at method X. If the program hits any breakpoint at X, then a violation of the Never call X from Y rule will be reported to the user. When method Y completes, the breakpoint on method X is removed.

[0022] The Do not call X from Y more than Z times pattern detector will also place a breakpoint at Y. If the Y breakpoint is hit in the course of program execution, an additional breakpoint will be added at method X. If the breakpoint to X is hit more than Z times, then a violation of the Do not call X from y more than Z times will be reported to the user, and the breakpoint at X will be removed. When method Y completes, the breakpoint on method X is removed.

[0023] The Do not store objects that are greater than size Z in X through method Y pattern detector will place a breakpoint in method Y that allows an object to be stored in S. Each store is then checked against size Z. Any store above size Z will be reported as a violation to the user.

[0024] To achieve the desired performance and scalability, preferably the tool has the following features.

[0025] First, the tool uses a cascading system of breakpoints that are applied to a running program to enforce a best practice pattern. Only the breakpoints necessary for detecting the "next" step in a pattern are active at any given time. This is advantageous both to control the number of breakpoints that need to be active at any given time and to minimize the overhead of running an application in this environment. Specifically, if the beginning elements of a pattern are never encountered within the program execution, this technique ensures that the execution of a program is not slowed due to the setting of breakpoints for later elements of the pattern.

[0026] Second, once the later elements of a pattern are detected, and a violation is flagged, then the breakpoint is removed dynamically. An example of this is where X is called from Y more than Z times. In this case, the breakpoint at X may be removed, because the pattern detector has reached the exit condition. Again, this has beneficial effects on performance such that breakpoints that are no longer needed for pattern detection are removed dynamically during program execution.

[0027] While it is apparent that the invention herein disclosed is well calculated to fulfill the objects stated above, it will be appreciated that numerous modifications and embodiments may be devised by those skilled in the art, and it is intended that the appended claims cover all such modifications and embodiments as fall within the true spirit and scope of the present invention.